A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening

Alberto Magni
University of Edinburgh
a.magni@sms.ed.ac.uk

Christophe Dubach
University of Edinburgh
christophe.dubach@ed.ac.uk

Michael F.P. O'Boyle
University of Edinburgh
mob@inf.ed.ac.uk

ABSTRACT

OpenCL has become the de-facto data parallel programming model for parallel devices in today's high-performance supercomputers. OpenCL was designed with the goal of guaranteeing program portability across hardware from different vendors. However, achieving good performance is hard, requiring manual tuning of the program and expert knowledge of each target device.

In this paper we consider a data parallel compiler transformation — *thread-coarsening* — and evaluate its effects across a range of devices by developing a source-to-source OpenCL compiler based on LLVM. We thoroughly evaluate this transformation on 17 benchmarks and five platforms with different coarsening parameters giving over 43,000 different experiments. We achieve speedups over 9x on individual applications and average speedups ranging from 1.15x on the Nvidia Kepler GPU to 1.50x on the AMD Cypress GPU. Finally, we use statistical regression to analyse and explain program performance in terms of hardware-based performance counters.

Categories and Subject Descriptors

D.1.2 [**Processors**]: Compilers, Optimization

General Terms

Experimentation, Measurement, Performance

Keywords

GPU, OpenCL, Thread coarsening, Regression trees

1. INTRODUCTION

GPUs are the corner-stone of modern high performance computing. The reason for their popularity is clear: they provide high performance within a manageable power budget. However, heterogeneous platforms supporting different programming models are difficult to program. OpenCL has been proposed as a common standard for heterogeneous devices and is now supported by all hardware manufacturers. OpenCL provides functional portability across a diverse range of platforms. The programmer writes explicit parallel kernels which are then compiled for the target device. Although *functionally* portable OpenCL is a relatively low-level language and it is far from being *performance* portable. OpenCL programs designed and tuned for one system are unlikely to perform as well on another platform. This problem is particularly acute given the diversity of modern GPU architectures. Even hardware from the same vendor might show very different characteristics from one generation to the next. AMD, for example, has recently changed their top-tier discrete GPUs from VLIW to SIMD cores [5]. OpenCL provides functional portability at the expense of a future of never-ending performance porting.

Ideally, HPC expert programmers would like analysis tools to help port and remap their applications to new parallel platforms thus reducing the effort of upgrading. However, performance varies widely across devices and it is often hard to fully understand the various factors affecting execution time. It is critical to discover, analyze and explain these factors to achieve maximum performance.

Thread-coarsening [24, 27] is known to have a significant impact on kernel performance as it directly affects how data parallel work is mapped to the underlying hardware. In the paper we examine the impact of thread-coarsening across architectures. We analyse the results in terms of hardware characteristics in a systematic approach based on regression. This analysis explains program behavior aiding the development of portable transformations.

To explore the impact of coarsening across different vendor platforms we have developed a thread-coarsening transformation within a source-to-source OpenCL compiler based on LLVM [2]. To understand how thread-coarsening affects performance we employ a statistical methodology based on regression trees. We profile the applications gathering platform-specific counters and use a platform-independent modeling technique to explain behavior. This methodology is extremely flexible and portable across processors, compilers and runtime libraries. It gives an intuitive and visual explanation of what are the relevant hardware features and how they relate to performance. Our regression-tree based methodology is not coarsening-specific; it can analyze any large set of profiler counters with the goal of identifying the ones that really matter for performance.

To summarize, this paper makes the following contributions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC '13, November 17 - 21 2013, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 $15.00.
http://dx.doi.org/10.1145/2503210.2503268

```
1 kernel void matrixTransposition(
2   int width, height, global float* in,out)
3 {
4   uint column = get_global_id(0);
5   uint row = get_global_id(1);
6
7
8   uint inputIdx = row * width + column;
9
10  uint outputIdx = height * column + row;
11
12
13  out[outputIdx] = in[inputIdx];
14 }
```

(a) Original code

```
1 kernel void matrixTranspositionCoarsened2x(
2   int width, height, global float* in, out)
3 {
4   uint column = get_global_id(0);
5   uint row0 = 2 * get_global_id(1) + 0;
6 ▶ uint row1 = 2 * get_global_id(1) + 1;
7
8   uint inputIdx0 = row0 * width + column;
9 ▶ uint inputIdx1 = row1 * width + column;
10  uint outputIdx0 = height * column + row0;
11 ▶ uint outputIdx1 = height * column + row1;
12
13  out[outputIdx0] = in[inputIdx0];
14 ▶ out[outputIdx1] = in[inputIdx1]; }
```

(b) After coarsening with factor 2.

Figure 1. Applying coarsening by factor 2 to `matrixTransposition` (`mt`).

1. New **thread-coarsening transformation** that deals with control flow divergence, synchronization and thread-remapping;
2. First comprehensive **evaluation of a fully portable LLVM-based compiler for OpenCL**;
3. **New methodology based on regression trees to identifies key performance features** affected by thread-coarsening.

2. MOTIVATION

This section introduces the OpenCL programming model, the thread-coarsening transformation and motivates the work presented in this paper.

2.1 OpenCL

OpenCL is now the standard programming model for parallel hardware such as CPUs or GPUs. The OpenCL host code manages the device and the kernel function which defines the work carried out by each data-parallel thread. When running an application the OpenCL runtime compiles the kernel into accelerator-specific assembly code, instantiates a fixed number of threads (*work-items*) and schedules them onto the multicore device.

Work-items are arranged on a multi-dimensional grid and to each one is assigned a unique identifier: the *thread_id*. Work-items are scheduled onto single cores in *work-groups* whose size is specified at kernel launch time.Work-items within a work-group can share data in local memory and synchronize using a barrier function.The programmer must decide how many *work-items* to instantiate, how much work each *work-item* carries out and the work-group size. These parameters have a critical effect on performance and it requires intimate knowledge of the target device to choose the optimal ones.

2.2 Thread Coarsening

We now introduce the thread-coarsening transformation with an example. Figure 1a shows the OpenCL kernel code for `matrixTransposition`. This is a two-dimensional kernel with one *work-item* associated with each element of the matrix to transpose. As we see the *get_ global_id* function retrieves the thread id along each dimension.

The coarsening transformation works by merging several *work-items* together. Figure 1b shows the effect of coarsening along the second dimension with a factor of two: each pair of consecutive *work-items* are merged into one. The steps necessary to coarsen this program are: (1) replicate each instruction that depends on the thread id along the second dimension (underscored in figure 1a), (2) transform the global id with an affine function to reflect the merging of *work-items* and (3) halve the number of *work-items* along the second dimension (not shown here since this is done in the host code).

Coarsening can improve performance reducing the amount of redundant computation. However, a major issue consists in choosing the optimal coarsening factor that leads to the best performance. Furthermore, correct coarsening is more difficult in kernels with non-trivial control-flow. For these reasons, we employ a source-to-source OpenCL compiler: it transforms the code automatically making possible a quick search of the transformation space.

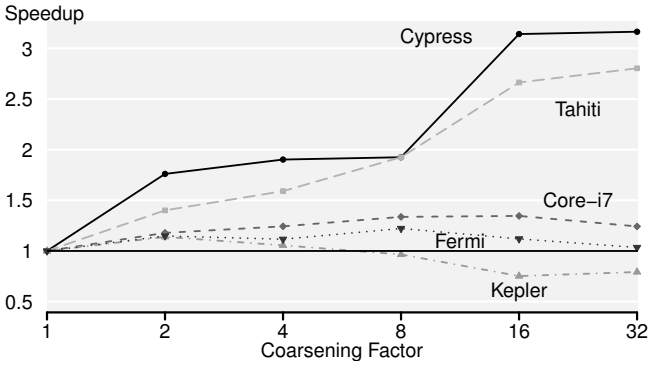


Figure 2. Speedup achieved by different coarsening factors on `matrixTransposition` on five platforms.

2.3 Performance Impact

We now look at experimental data gathered on five different platforms for our `matrix transposition` example. The platforms are two AMD GPUs (Cypress and Tahiti), two Nvidia GPUs (Fermi and Kepler) and one Intel CPU (Core-i7). Figure 2 shows for each of these platforms the relative performance achieved for different coarsening factors with

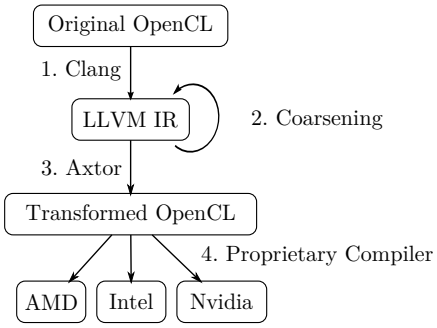


Figure 3. Compiler toolchain

a baseline of 1 (*i.e.*, no coarsening). As can be seen the impact of the coarsening varies widely depending on the device. For the two AMD GPUs, we achieve the highest speedups around 3x with coarsening factor of 32. On the Nvidia Fermi GPU the optimal is achieved with coarsening factor 8 and leads to about 25% improvement. Nvidia Kepler GPU achieves a 20% improvement with a coarsening factor of 2; performance deteriorates with larger coarsening factors. Finally, on the CPU we record a speedup of 1.4x for a coarsening factor of 16.
This simple example illustrates the challenges encountered by programmers when porting OpenCL programs. Choosing a single coarsening factor for all the devices would lead to suboptimal performance even among devices from the same vendor. In section 7 we provide an explanation for the performance differences we record on our devices.

3. SRC-TO-SRC OPENCL COMPILER

Extensive cross-architectural evaluation of thread-coarsening is only possible with the support of a source-to-source compiler. Our compiler toolchain is depicted in figure 3. The original OpenCL C source code is translated to LLVM bitcode using the `clang` open-source C front-end (stage 1). We then apply the coarsening pass described in the next section, at the LLVM bitcode level (stage 2). The transformed LLVM program is then translated back to OpenCL C using the `axtor` [19] LLVM OpenCL-backend (stage 3). The transformed OpenCL program is then fed into the hardware vendor proprietary compiler (stage 4) for code generation. This is the first compiler toolchain that enables the evaluation of IR-transformations on multiple parallel devices without support for specific LLVM-backends. Previous source-to-source approaches such as [27] working on the AST do not provide the necessary flexibility to evaluate complex transformations on real applications.

4. COARSENING IMPLEMENTATION

Thread coarsening has been implemented as a *function-pass* working on LLVM IR. It increases the amount of useful work carried out by a single thread replicating the instructions of the kernel body and reducing the overall number of threads. Only the instructions that differ across threads need to be replicated. For this reason we first perform divergence analysis before coarsening is applied.

```
1: function FindDivergentInstructions
2:     seeds=get_global_id ∪ get_local_id
3:     currentInsts = divergentInsts = seeds
4:     while currentInsts ≠ ∅ do
5:         newInsts = ∅
6:         for all inst in currentInsts do
7:             users = forwardSlicing(inst)
8:             divergentInsts = divergentInsts ∪ users
9:             newInsts = newInsts ∪ users
10:        currentInsts = newInsts
11:    return divergentInsts
```

Figure 4. Divergence analysis pseudocode

```
1: function Coarsening(coarseningFactor)
2:     allDivergentInsts = FindDivergentInstructions()
3:     divergentBranches = getBranches(divergentInsts)
4:     divergentRegions =
5:         getDivergentRegions(divergentBranches)
6:     for all inst in divergentInsts do
7:         for subId in [1 ... coarseningFactor] do
8:             newInst=replicateInst(inst, subId)
9:             insertInst(newInst);
10:    for all region in divergentRegions do
11:        for subId in [1 ... coarseningFactor] do
12:            newRegion=replicateRegion(inst, subId)
13:            insertRegion(newRegion);
```

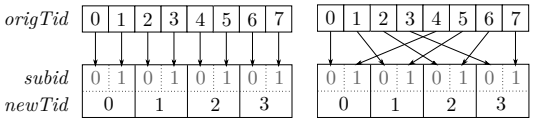
Figure 5. Thread coarsening pseudocode

4.1 Divergence analysis

A requirement for effective thread-coarsening is the identification of *divergent* instructions: the ones that behave differently in different threads. An instruction is *non-divergent* if it is executed by all the threads and gives the same output for all threads. Divergence analysis [8] finds all the instructions that are control- and data-dependent on the *thread_id* since the thread_id is the only feature distinguishing threads in the OpenCL programming model. As an example consider the code in figure 1a. Here the divergent variables are underlined. As we are coarsening along the second dimension (rows) only the `row` variable diverges and needs replication, as do all other instructions referring to `row`.
Figure 4 gives an high level view of the algorithm we implemented in LLVM. We first look for all the instructions data-dependent on the *thread_id* (either local or global) marking them as divergent. Data-dependency is checked using forward code slicing [25]. The newly found divergent instructions are then used as seeds for further steps of slicing until no new divergent instructions are found.

4.2 Thread Coarsening

Once that divergent instructions have been identified we can finally apply the coarsening pass. The pass replicates all divergent instructions by the coarsening factor. Divergent LLVM-bitcode instructions are cloned and inserted right after the original ones. Considering figure 1b the coarsened instructions are identified by the symbol ▶.
The newly inserted instructions will work on the data points referenced by the threads before the merging in the original iteration space as shown in figure 6. The N threads in the original iteration space are identified by $origTid$, where $origTid \in [0, N-1]$. The new iteration space resulting from coarsening with factor cf is $newTid \times subId$ where $newTid \in [0, (N-1)/cf]$ and $subid \in [0, cf-1]$ is the sub-



(a) Naive coarsening **(b)** Coarsening with stride 4

Figure 6. Two different thread-coarsening mappings. *origTid*, *subid* and *newTid* refer to the notation in section 4.2

index of the thread being merged. Coarsening is defined by the following transformation:

$$origTid \mapsto newTid \times subid \tag{1}$$

with constraint

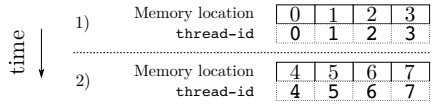
$$newTid \cdot cf + subid = origTid. \tag{2}$$

The pseudo-code of our transformation is given in figure 5. Notice the function `replicateInst` at line 8: it clones the given instruction modifying the operands of the new one so to make it work on the specified *subid*. Special care must be taken in the presence of control flow. If a branch is *non-divergent* (i.e. it is taken or not taken by all the threads) then it can be safely maintained as it is: coarsening will replicate the divergent instructions in the region controlled by the branch without modifying the CFG. If instead the branch is divergent such approach is unsafe and leads to corruption of the semantics. We manage such cases treating the whole CFG region controlled by a divergent branch as a single statement. All the basic blocks enclosed by the branch and its immediate post-dominator are called a *divergent region*. During coarsening divergent regions are cloned and placed right after the original ones. Divergent regions are identified starting from varying branches (figure 5, line 5) and replicated along with divergent instructions (line 12). Region cloning is unsafe in the presence of synchronization (`barrier` function call) inside regions: this breaks the program semantics. Our transformation instead is always safe since we clone only divergent regions. Here we exploit the property that the OpenCL specification ensures that barriers can only be placed in regions which behave in non-divergent fashion. Regions with barriers are therefore always treated as non-divergent.

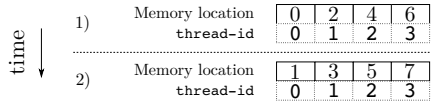
4.3 The stride option

All the GPUs considered in this work have a warp-based execution model. A warp is a set of consecutive hardware threads executed in lock-step. On GPUs the best memory access pattern is achieved when consecutive work-items access consecutive memory locations: coalesced access. After coarsening programs might loose this property due to thread remapping. For this reason, we propose an extension to basic coarsening which restores the original coalesced access pattern. We merge together threads non-consecutive in the original space but separated by a stride. The new constraint on the thread iteration space transformation from formula 1 becomes:

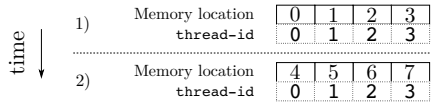
$$\lfloor \frac{newTid}{st} \rfloor \cdot cf \cdot st + newTid \cdot \bmod(st) + subid \cdot st = origTid \tag{3}$$



(a) Original memory accesses for `x = A[thread_id]`. Threads are dividend in two warps with consecutive threads accessing consecutive memory locations



(b) Memory accesses for the coarsened version of the load: `x1 = A[2 * thread_id]; x2 = A[2 * thread_id + 1];`. Accesses are no more coalesced.



(c) Recovered coalesced access pattern using a stride of 4: `x1 = A[thread_id % 4]; x2 = [thread_id % 4 + 4];`.

Figure 7. Three different memory access patterns from an originally coalesced load. 7a no coarsening, 7b coarsening without stride, 7c coarsening with stride

Parameter	Possible values
Coarsening factor	{1, 2, 4, 8, 16, 32}
Coarsening dimension	{0, 1}
Stride	{1, 2, 4, 8, 16, 32}
Local work group size (for each dimension)	{*dev min*, ..., 32, 64, 128, ..., *dev max*}

Table 1. Parameter space

where *st* is the stride. The effect of stride on the remapping is showed in figure 6b. Figure 7 shows the memory locations accessed by 8 threads divided into 2 warps when performing a coalesced load (`x = A[thread_id]`) in 3 different cases: no coarsening (7a), coarsening by factor 2 (7b), and coarsening with stride 4 (7c). Strided coarsening is beneficial for performance as shown in section 6.

5. EXPERIMENTAL SETUP

This section introduces the compiler options controlling coarsening, the benchmarks and the devices used in the experiments.

5.1 Compiler Parameter Space

The parameter space we considered is shown in Table 1. We have three parameters controlling coarsening: factor, dimension and stride. In the case of two-dimensional benchmarks, the second parameter allows us to control along which dimension coarsening is applied. The last parameter, local work group size, determines the total number of threads, along each dimension, in a work-group. This maximum value is constrained by the device limitations on a per-kernel basis. We have exhaustively explored all combination of our parameters leading to about 150-300 configurations for one dimensional kernels and 1000-2000 configuration for two dimensional kernels depending on the device limitation. These sum up in about 43000 evaluated configurations across all programs and devices.

	Program name	Source	Num. threads
1)	binarySearch	AMD SDK	64K
2)	blacksholes	Nvidia SDK	256K
3)	convolution	AMD SDK	1M
4)	dwtHaar1D	AMD SDK	2M
5)	fastWalshTrans	AMD SDK	512K
6)	floydWarshall	AMD SDK	(6K x 6K)
7)	mriQ	Parboil	256K
8)	mt	Nvidia SDK	(4K x 4K)
9)	mtLocal	Nvidia SDK	(4K x 4K)
10)	mvCoal	Nvidia SDK	256K
11)	mvUncoal	Nvidia SDK	256K
12)	nbody	AMD SDK	64K
13)	reduce	AMD SDK	8M
14)	sgemm	Parboil	(512 x 512)
15)	sobel	AMD SDK	(512 x 512)
16)	spmv	Parboil	16K
17)	stencil	Parboil	(512 x 510 x 62)

Table 2. OpenCL applications

5.2 Benchmarks

We have used 17 benchmarks from various sources as shown in Table 2. In the case of *Parboil* benchmarks we used the *opencl_base* version. The table also shows the global size (total number of threads) used. We made sure that for each benchmark we are able to change the local-work-group size at will. The baseline performance reported in the results section is that of the best work group size as opposed to the default one chose by the programmer. This is necessary to give a fair comparison across platforms when the original benchmark is written specifically for one platform. All the performance numbers we report in section 6 are relative to the kernel-only execution time of the original un-coarsened code after being transformed by the `axtor` OpenCL-backend. Each experiment has been repeated 15 times for GPUs and 31 times for the CPU aggregating the results using the median.

5.3 Devices

For our experiments we used five devices from three different vendors. From Nvidia we used GTX480 and Tesla K20c. The first one is based on the Fermi architecture [4] with 15 streaming multiprocessors (SM) and 32 CUDA cores for each. Tesla K20c instead ships the latest Kepler architecture [6]. K20c has 13 SMs each with 192 CUDA cores.
From AMD we used Radeon HD 5900 and Tahiti 7970. The first one, codenamed Cypress, has 20 SIMD cores each with 16 thread processors, a 5-way VLIW core. The Graphics Core Next architecture in Tahiti 7970 is a radical change for AMD. Each of the 32 computing cores contains 4 vector units (of 16 lanes each) operating in SIMD mode. This exploits the advantages of dynamic scheduling as opposed to the static scheduling needed by VLIW cores.
The last device we looked at is a classic multicore CPU: Intel i7-3820 with Sandy-Bridge architecture, it mounts four cores each with 2 hardware threads.

5.4 Profiling

To understand the effects of thread-coarsening on performance collect profiling information specific for each architecture. For the AMD and Nvidia GPUs we have used the available proprietary OpenCL profiler from each vendor [1, 3]. For the Intel CPU we instrumented the code kernel code using PIN [18] extracting architecture independent features [13] using the tool MICA [7]. This approach offers a rich set of features although not device specific. Using these tools, we collected counters about the number of executed instructions, instruction mix, cache and memory behaviour.

6. RESULTS

This section presents the performance improvements achieved when using coarsening.

6.1 Speedups

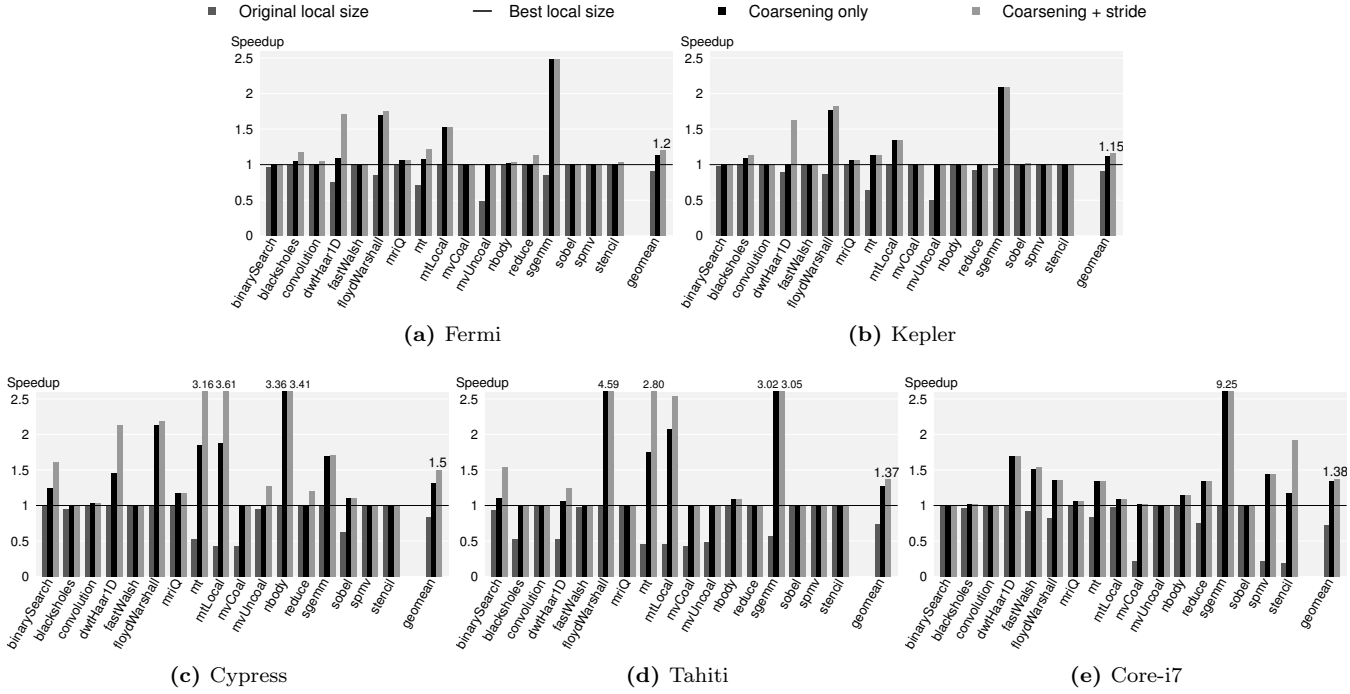
Figure 8 shows the maximum speedup obtained across all the transformation settings (see section 5.1) for each program and device. We fixed the baseline performance to that of the original program (coarsening factor 1) with its best local work group size. This is a strict baseline and ensures a fair comparison across devices. In the figure the baseline is represented by the horizontal line at 1.0. The first bar reports the performance of the program run with the *original* local work group size. This often proves to be less than 1.0 because of the need to tune this parameter for each platform specifically as showed in [17].
The second bar shows the best speedup achievable when the application is *coarsened* and the third one shows the additional gain from applying the *stride* thread-remapping. Coarsening ensures significant speedups for most of the benchmarks across the five devices. Speedups range from a few percent up to 9.25 for `sgemm` on the Core-i7.
Platforms. The two Nvidia GPUs (8a, b) show very similar performance. The average improvement is 1.15x for Kepler and 1.2x for Fermi. The programs that improve the most after coarsening are `dwtHaar1D`, `floydWarshall`, `mt`, `mtLocal` and `sgemm`. Other kernels do not show significant improvements, in particular on Nvidia Kepler. This proves the need to thoroughly study performance so to select the compiler transformation on a per-program basis. The two GPUs by AMD (8c, d) instead show more significant and diversified speedups averaging 1.5 for Cypress and 1.37 for AMD Tahiti. Differences in the architectures design are reflected in the speedup differences for benchmarks such as `dwtHaar1D`, `nbody` and `sgemm`. Finally on the Core-i7 coarsening gives a speedup of 1.38 on average. Applications such as `spmv` and `stencil` which do not improve on the GPUs do improve on the CPU.
Stride. The stride optimization is particularly effective for AMD GPUs giving an average improvement of 1.14 and 1.07 over the best coarsening-only configuration for AMD Cypress and AMD Tahiti respectively. On Nvidia and Intel devices the speedup given by the stride option is less consistent: on average 1.05 on Nvidia Fermi and 1.03 on both Nvidia Kepler and Intel Core-i7. On Nvidia applications such as `dwtHaar1D`, `mt` and `reduce` show significant speedups up to 1.56. The stride values giving the best performance are the largest ones in our space (16 and 32). This is because in both the Nvidia and AMD devices 16 consecutive hardware threads need to access consecutive memory locations to achieve a coalesced memory transaction.
A detailed explanation of the overall behavior of the benchmarks is given in section 7.

6.2 Effect of Coarsening Factor

We now consider the effect of the coarsening factor on performance. Figure 9 shows the maximum achievable speedup

Type	Name	Model	GPU Driver	OpenCL version	Linux kernel
CPU	Core-i7	Intel i7-3820	None	1.1 Intel OpenCL SDK	3.1.10
GPU	Cypress	AMD Radeon HD 5900	1084.4	1.2 AMD-APP 1084.4	3.1.10
GPU	Tahiti	AMD Tahiti 7970	1084.4	1.2 AMD-APP 1084.4	3.1.10
GPU	Fermi	Nvidia GTX 480	304.54	1.1 CUDA 5.0.1	3.2.0
GPU	Kepler	Nvidia K20c	304.54	1.1 CUDA 5.0.1	3.1.10

Table 3. OpenCL devices used for experiments.



(a) Fermi **(b)** Kepler

(c) Cypress **(d)** Tahiti **(e)** Core-i7

Figure 8. Speedup given by the best coarsening configuration. The first bar (■) represents the performance of the application run with the local work group size defined by the benchmark suite. The second one (■) is the speedup achieved by the best coarsening configuration. The third bar (■) represents the performance of the best coarsening plus stride configuration.

for each coarsening factor when setting the local work group size and stride to their best values, for all programs and architecures. This overall view shows the difficulties of tuning the right coarsening factor. The speedup curves of a single application look very different across the five devices. For example `binarySearch` (first row) is not sensitive to any coarsening factor on Nvidia Fermi, Kepler and Intel Core-i7, while it shows significant improvements on AMD Cypress and Tahiti. Furthermore, the optimal coarsening factor is 2 for Cypress and 8 for Tahiti. `spmv` and `stencil` (last 2 rows) experience slowdowns on the GPUs while coarsening by factor 32 gives the best performance on the CPU. The technique we introduce in the next section manages automatically this large amount of information providing useful indication on what counters are revelant for performance.

7. PERFORMANCE ANALYSIS

This section studies the effects of thread-coarsening on performance using performance counters and regression trees.

7.1 Regression Trees

Classical performance evaluation methods consist in building analytical models [22]. Such approaches fall short when dealing with several architectures. Developing an analytical model is infeasible due to the large variety of hardware and runtime libraries taken into account. Instead, to analyze the effects of our transformations we propose using counters cupled with regression trees [20]. Regression trees are a robust statistical analysis and offers two unique advantages: (1) it is *platform-portable* since creating the performance model is fully automated and requires no human effort; (2) the resulting model is *easy to visualize and understand*, and as such offers insights into the factors that affects performance.

7.2 Problem Formulation

Let $s_{p,cf}$ be the speedup achieved on program p with coarsening factor $cf \in \{1, 2, 4, 8, 16, 32\}$ and the best possible stride and local work group size. The speedup are expressed for each program with respect to the un-coarsened version, which means $s_{p,1} = 1 \ \forall p$. Let $\vec{f}_{p,cf}$ be the vector of performance counters extracted from the same run.

We want to find a mapping between performance counters $\vec{f}_{p,cf}$ and speedup $s_{p,cf}$ to understand the factors affecting performance. To achieve this, we record the execution time and performance counters for all coarsening factors on all our programs for each platform.

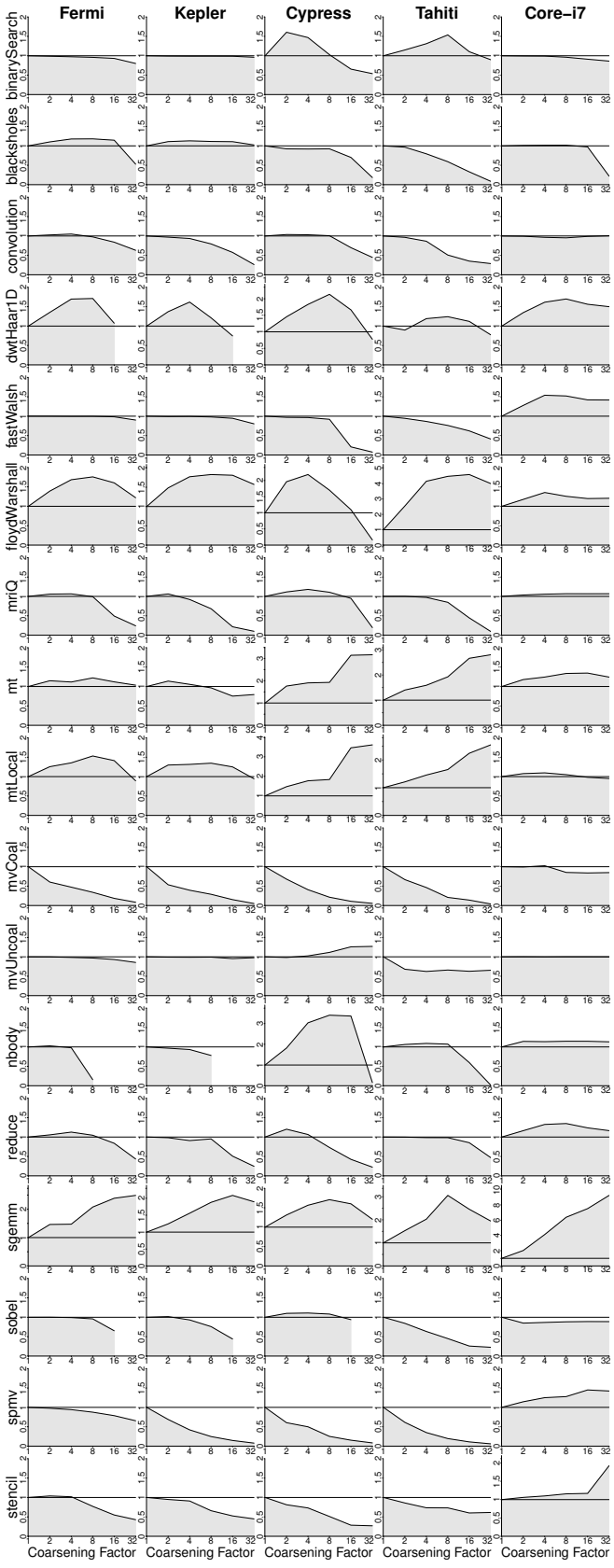


Figure 9. Maximum attainable speedup for all the programs and devices as a function of the coarsening factor.

The regression trees describe the mapping $\vec{f}_{p,cf} \rightarrow s_{p,cf}$ acting as a broad model for the speedups in figure 8. Once constructed, the trees can then be used to explain the observed behavior. Tree construction involves recursively partitioning the data into sets with similar performance $s_{p,cf}$ based on the performance counters $\vec{f}_{p,cf}$. To increase the accuracy of the model we used in the construction of the trees the top best 5 performing configurations for each coarsening factor. We have used the R package *tree* (version 1.0-33) using deviance as an impurity metric. The trees are then pruned using cross-validation ensuring that the expected speedup has a correlation of at least 0.8 with the curves showed in figure 9 and error about 10% for all the GPUs. These performance results are a little worse for Core-i7 (around 0.7 correlation) due to the usage of platform independent features. The resulting tree still provides an insightful explanation about performance differences.

The trees for each platform are shown in figures 10a–e. Each node represents the relative increase or decrease of a performance counter over its value for the baseline configuration. The right child of a node satisfies the condition of that node while the left one does not. Based on the inherent property of a regression tree, the conditions near the top of the tree are the ones that are best at distinguishing bad from good performance. Note that for visualization and explanation purposes the trees have been further pruned manually.

7.3 Per-device analysis

7.3.1 Nvidia Fermi GPU

As can be seen in figure 10a, the number of memory loads is the most important counter to consider. A reduction in the number of loads leads to a significant performance improvement: 1.70x on average. This is the case for 2 applications: `sgemm` and `floydWarshall`. Coarsening gives an opportunity to reuse in a single thread values that were originally loaded by different threads, thus reducing the overall number of loads executed.

Going down one level, the second most important counter is the number of branches. Control flow is critical for GPU performance. If the output of a branch diverges within a warp all its threads will execute both paths of the branch thus leading to a lower core occupancy. Configurations in the lower right corner, where the number of branches has increased and the number of loads has not been reduced, show significant slowdowns. In this region fall `spmv`, `mvCoal` and `stencil` with slowdowns on average 0.4x. The bodies of these three kernels are enclosed in divergent regions which are entirely replicated by the coarsening pass leading to an increase in the relative number of branches hence lower occupation. Finally, the last counter selected concerns the cache utilization. As we show in figure 11 coarsening changes kernel memory behaviour. If the working set of the benchmark does not increase we experience a moderate improvement: 1.06x. On the other hand trashing the cache leads to major performance drops. More details and examples on this in section 7.4.3.

7.3.2 Nvidia Kepler GPU

As described in section 5.3 the Kepler architecture is an evolution of Fermi maintaining the same design principles. This explains the similarities between the two trees in figures 10a and 10b. The number of loads and branches are

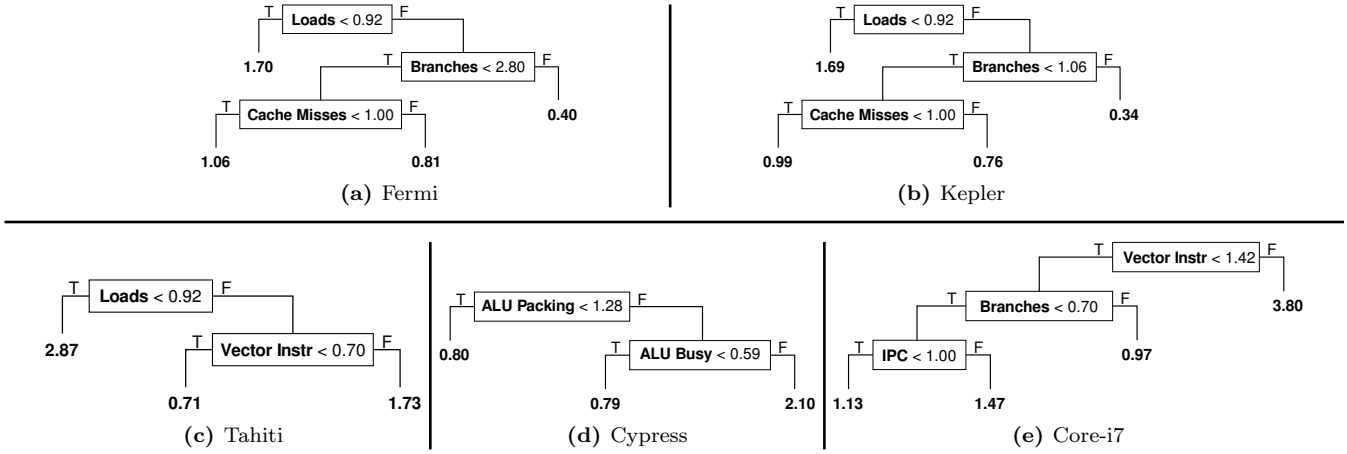


Figure 10. Regression trees for the five architectures. The labels of the nodes identify the profiler counter used to split the data. In each node the specified feature is compared against its relative increase over its value in the default configuration. The numbers in the laves represent the expected speedup of configurations falling the the specified partition.

still the most important counters. In this case though the tree shows higher sensitivity to branches. This is due to the change in the instruction scheduling policy from a dynamic one in Fermi to a static one in Kepler. While on Fermi a relative increase of 2.8 of the number of branches leads to a significant slowdown (0.4x of the original performance) this threshold is lowered to 1.06 for Kepler. Overall the expected speedups given by the tree reflect the lower performance improvement achievable on Kepler with respect to Fermi. The reason for this difference is the higher number of computing cores mounted on a Kepler streaming multiprocessor (192 vs 32) which ensures better performance to applications running a higher number of threads even if they execute redundant operations.

7.3.3 AMD Tahiti GPU

As seen on the Nvidia devices a reduction in the number of loads leads to a significant improvement on Tahiti too: 2.87x on average. Descending on the right branch of the root the second most important counter is related to the utilization of the vector unit mentioned in section 5.3. For benchmarks `mt`, `mtLocal` and `nbody` the increase in data-dependence-free instructions in a single thread allows a better utilization of the vector lanes of the GPU cores. On the other hand kernels like `mvCoal`, `mvUncoal`, `spmv`, `stencil` show a reduction in the vector utilization due to the divergent regions they contain. These limit the scope of vectorization reducing the utilization of the SIMD lanes.

7.3.4 AMD Cypress GPU

The cores of the Cypress GPU are based on a VLIW architecture. The *ALUPacking* counter measures the utilization of the 5-way VLIW cores and is the most important counter. The increase in the core efficiency is due to a better utilization of the functional units thanks to the better scheduling now performed on the larger number of instructions available after coarsening. Notice that the core utilization is more relevant for performance than a reduction in the number of loads which is not used to discriminate benchmarks on this device. On Cypress the importance of the efficient utilization of the computing cores is confirmed by the choice of *ALUBusy* as second most important counter. Sufficient utilization of the ALU leads to a 2.10x speedup.

7.3.5 Intel Core-i7 CPU

Finally, for the Core-I7 CPU the most important counter is the utilization of SIMD vector instruction. Coarsening gives more scope for the auto-vectorizer to find suitable instructions to pack together. The average speedup achievable thanks to an increase in the SSE utilization is 3.8x. The second most important counter concerns branches. A reduction in the number of executed branches leads to an improvement of 1.27x. The last counter measures the achievable instructions per cycle performance for the coarsened kernel. This counter is computed by MICA [7] calculating the cycle count for an ideal schedule of the instructions in the kernel. Coarsening gives the opportunity to increase the available instruction level parallelism thus increasing the IPC performance. This reflects in an average improvement of 1.47x.

7.4 Performance Counters

The previous analysis identifies what are the important hardware counters related to performance when applying coarsening. In this section, we now study how coarsening affects the counters selected by the trees and how these affect performance, while others such as cache misses are architecture specific. In figure 11 we plot the relative value of a performance counter as a function of the coarsening factor. Counters are normalized with respect to the value of the counter of the baseline configuration (no coarsening).

7.4.1 Memory Loads

We first look at the impact of coarsening on the number of memory loads in the program. This is platform independent counter but is very important for both Nvidia devices and Tahiti. Considering figure 11a we see that for `floydWarshall` (6) and `sgemm` (14) the number of loads is decreases as the coarsening factor increases. This is due to the reuse in registers of values originally loaded by different threads, thus leading to the elimination of the redundant loads. The possibility to reduce the number of loads is extremely ben-

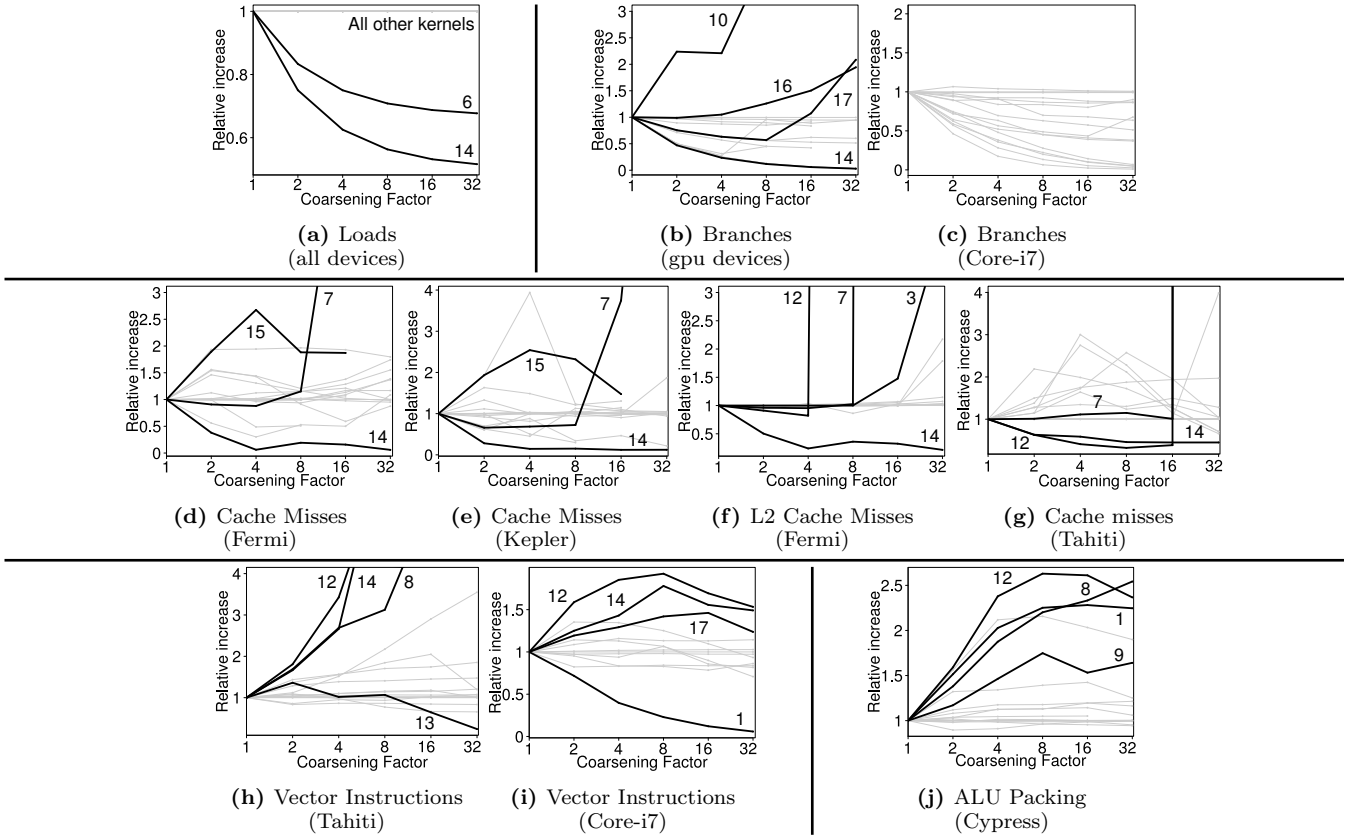


Figure 11. Relative value of hardware performance counters as a function of the coarsening factor. All values are normalized with respect to the value of the counter for coarsening factor 1. Each line represents a kernel. The thick black lines correspond to the benchmarks referenced in the text, the numbers beside them identify the kernels according to table 2.

eficial in GPUs given that these devices are usually limited by memory latency.

7.4.2 Branches

Figures 11b and 11c show the effect of coarsening on the total number of branches. The number of branches is similar across all the GPUs but differs on the CPU. This is due to the differences in the ISA and in the way in which branch instructions are classified by MICA [7]. Thread coarsening generally reduces the number of executed branches. due to the fact that our coarsening pass, relying on divergence analysis, preserves non-divergent branches replicating only the instructions they control. This leads to an overall reduction of the number of branches with respect to the total number of instructions, consider for example `sgemm` (14). Important exceptions to the trend are `mvCoal` (10), `spmv` (16) and `stencil` (17) which all show divergent regions as explained in section 7.3.1. All these benchmarks show large performance degradations when coarsening.

7.4.3 Cache Utilization

Figures 11d-g describe the cache utilization for the two Nvidia GPUs and Tahiti. As we can see from the general trend of the grey lines coarsening tends to put the cache under stress increasing the miss rate. This is due to an increase in the working size of each thread. Considering for example `sobel` (15) we see that an increase in the cache miss rate for both Fermi and Kepler: this justifies the performance drop we record in figure 9. Extreme cases are `mri-q` (7) and `nbody` (12) for which we see that coarsening by high factors (4 and 8 on Fermi, 16 on Tahiti) leads to a large increase in the number of misses which corresponds to significant performance penalties. The reason for this behavior lies in the structure of the code: both these kernels have a non-divergent loop in their body. The coarsening pass replicates the instructions within the loop body thus increasing the loop working-set. After a certain threshold the advantage given in the reduction of the number of branches is outweighed by the penalty due of cache misses. An exception to the general trend is `sgemm` (14) for which the possibility to save load instructions also has a beneficial effect on the cache utilization.

7.4.4 Vector Instructions

Figures 11h-11i show the utilization of vector instructions for AMD Tahiti and Intel x86 (SSE). These are the only architectures where vector utilization is significant. Figure 11h shows the utilization of vector instructions on Tahiti when coarsening. For applications like `sgemm` (14) and `mt` (8) coarsening has a beneficial effect on the vector lanes utilization leading to significant speedups. Looking at figure 11i we see that the applications that increase the most the vector utilization are `sgemm` (14), `stecil` (17) and `nbody` (12). For the first two kernels we experience relevant speedups for high coarsening factors. On the other hand a better uti-

lization of vector instructions does not lead to significant improvements for `nbody` due to a worse cache utilization, as described in section 7.4.3. Finally `binarySearch` shows a reduction in the utilization of vector instructions, this contributes in explaining the absence of improvements for this benchmarks (see the corresponding plot in figure 9).

7.4.5 ALU Packing

The degree of utilization of the 5-way VLIW cores for the Cypress architecture is showed in figure 11j. Applications such as `nbody` (12), `mt` (8), `binarySeach` (1) and `mtLocal` (9) show an increase in the utilization of the cores. This means that the AMD compiler is able to better schedule the larger amount of data-dependency-free instructions available after coarsening. As we can see from figure 9 the listed applications greatly benefit from coarsening confirming the importance of improving the utilization of the arithmetic cores when tuning programs for Cypress.

Summarizing the results of our analysis we can conclude that coarsening works the best by reducing redundant computation. On GPUs the possibility to save loads and branches has proven to be very effective for performance. Attention must be placed in the kernel cache utilization since coarsening usually increase the working set size. Architectures from AMD and Intel enjoy a better utilization of the vector units after coarsening thanks to the larger number of data-dependency free instructions now available to the compiler for scheduling.

7.5 Towards thread-coarsening tuning

The analysis results given by the regression trees represent the first fundamental step in understanding the problem of coarsening. As seen, the trees are successful in the identification of the device-specific hardware and software characteristic that affect performance. This information can be effectively used to build accurate tuning euristics. For example, on Nvidia, static program characteristics like the number of divergent loads and branches can be used to evaluate the profitability of coarsening, given that high values of these features lead to bad performance. Finer tuning can also take into account cache utilization using static estimation of the working set size [26] of the new program version. Both AMD GPUs and the Intel CPU benefit from high ILP; static approximation of this metric [22] can be used to determine the profitability of coarsening.

8. RELATED WORK

Thread coarsening and divergence analysis. Thread coarsening for GPU computing has been first proposed by Volkov *et al.* [24] as an hand optimization to improve linear algebra applications. Unkule *et al.* [23] propose a first evaluation of coarsening. The transformation has only been applied to a very limited set of CUDA applications. Experiments have been deployed only on one device without providing any explanation of the performance results.
Divergence analysis for parallel languages has been proposed by Coutinho *et al.* [8]. They characterize applications based on the amount of divergent instructions helping manual tuning and enabling peephole optimizations to reduce branching overhead. Karrenberg *et al.* [15] use divergence analysis as a tool to retain uniform control flow when applying whole function vectorization [14].
Transformations for parallel languages. Ryoo *et al.* [21] have been the first to propose compiler transformations for GPGPU computing. They address memory related optimizations with the goal of improving access patterns. Liu *et al.* [17] perform an analysis of input sensitivity of CUDA programs choosing the best block size according the input size. Zhang *et al.* [28] propose a dynamic thread remapping policy to reduce the impact on performance of branch divergence and non-coalesced accesses. High level program tuning for GPUs has been the target of extensive research too. Sponge [12] is a compiler toolchain for the optimization of streaming programs for GPUs. Dubach *et al.* [10] propose a set compiler optimizations to improve performance of OpenCL programs generated by the Lime Java compiler. Cross-architectural memory access pattern transformations for OpenCL is the research topic of Grewe *et al.* [11].
The first proposal of a cross-architectural compiler for OpenCL is due to Yang *et al.* [27]. They evaluate an extensive set of code transformations on two Nvidia and one AMD GPUs. Their source-to-source transformations have been implemented in the Cetus compiler working on the AST. This limits the applicability of their methodology to just simple linear algebra programs. Our approach of applying transformations at the LLVM-IR level is instead much more powerful and flexible. Lastly they do not provide an explanation for the different reactions they obtain from different devices to the same code transformation.
Performance Modelling. Sim *et al.* [22] propose an analytical model for runtime prediction and bottleneck identification. The model has been developed with extensive hand work to fit a particular Nvidia GPU model. This is a major weakness in an environment subject to rapid changes both in hardware and in software such as the one of GPUs. A more agile performance characterization method like ours is therefore much more suitable for this context. A run-time prediction model based on linear regression has been proposed by Kerr *et al.* [16]. The strength of this work lies in the usage of platform-independent feature collected using Ocelot [9]. The focus of this work is on accurate execution time prediction rather than compiler optimization analysis.

9. CONCLUSION AND FUTURE WORK

In this work we presented the first fully portable OpenCL compiler tool-chain aimed at the evaluation of LLVM-based compiler transformations. We performed an extensive performance study of our implementation of the thread-coarsening compiler transformation. Our experimens have led to the evaluation of 43000 coarsening configurations across five devices by three vendors. The average performance improvement is 1.20x on Nvidia GTX480, 1.15x on Nvidia Tesla K20c, 1.37x on AMD Tahiti 7970, 1.50x on AMD Radeon HD 5900 and 1.38x on Intel i7.
The effects of coarsening on run-time performance have been studied by a statistical approach based on regression trees. The methodology has proven to be affective on identifying the hardware features relevant for performance for the five devices taken into consideration.
Building on top of the insights gained using the proposed data analysis technique future work will be directed to guide the program tuning process developing a methodology to choose the best coarsening configurations across different OpenCL devices.

10. REFERENCES

[1] AMD Inc., AMD APP Profiler `http://developer.amd.com/tools/heterogeneous-computing/amd-app-profiler/`.

[2] The llvm compiler infrastructure `http://llvm.org`.

[3] NVIDIA Corporation, NVIDIA Profiler `http://docs.nvidia.com/cuda/profiler-users-guide/`.

[4] Nvidia's Next Generation CUDA Compute Architecture: Fermi `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[5] *AMD Accelerated parallel processing OpenCL*, 2012.

[6] Nvidia's Next Generation CUDA Compute Architecture: Kepler `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012.

[7] MICA: Microarchitecture-Independent Characterization of Applications `http://boegel.kejo.be/ELIS/mica/`, 2013.

[8] B. Coutinho, D. Sampaio, F. Pereira, and W. Meira. Divergence analysis and optimizations. PACT, pages 320 –329, oct. 2011.

[9] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[10] C. Dubach, P. Cheng, R. M. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *PLDI*, pages 1–12, 2012.

[11] D. Grewe, Z. Wang, and M. F. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. CGO '13. ACM, 2013.

[12] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. ASPLOS '11, pages 381–392, New York, NY, USA, 2011. ACM.

[13] K. Hoste and L. Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. pages 83 –92, oct. 2006.

[14] R. Karrenberg and S. Hack. Whole-function vectorization. CGO '11, pages 141 –150, april 2011.

[15] R. Karrenberg and S. Hack. Improving performance of opencl on cpus. CC, pages 1–20, 2012.

[16] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling gpu-cpu workloads and systems. GPGPU '10, pages 31–42, New York, NY, USA, 2010. ACM.

[17] Y. Liu, E. Zhang, and X. Shen. A cross-input adaptive framework for gpu program optimizations. IPDPS '09, pages 1 –10, may 2009.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[19] S. Moll. Decompilation of LLVM IR, 2011.

[20] B. D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, Jan. 1996.

[21] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[22] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. PPoPP '12, pages 11–22, New York, NY, USA, 2012. ACM.

[23] S. Unkule, C. Shaltz, and A. Qasem. Automatic restructuring of gpu kernels for exploiting inter-thread data locality. CC, pages 21–40, 2012.

[24] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[25] M. Weiser. Program slicing. ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[26] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 350–360, 2011.

[27] Y. Yang, P. Xiang, J. Kong, M. Mantor, and H. Zhou. A unified optimizing compiler framework for different gpgpu architectures. *TACO*, 9(2):9, 2012.

[28] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. ASPLOS '11, pages 369–380, New York, NY, USA, 2011. ACM.